Source Code Documentation for LIS Job Management
Submitted for LIS Milestone-G

Contents

1 Description

The parallel processing management system plays a critical role to connect the land surface modeling job to the underlying multi-processor parallel computing hardware platform, in our case, the LIS Beowulf cluster, to achieve the goal of near real-time processing of high-resolution land surface simulation. The system implements the task-pool paradigm, with a request-response model for job distribution. This design proves to be highly reliable, efficient and scalable.

1.1 Master node job processing

We use a modified version of the "pool of tasks" scheme for the parallel processing of the global simulation. A pool of tasks paradigm is equivalent to a master - slave programming model, where a single processor will act as a master node and distribute jobs to the slave (compute) nodes. In the LIS "pool of tasks" design, we use one of the IO nodes as the master node. We refer to the master node as "farmer" and the compute nodes as "dogs".
The master node ("farmer") will keep three tables on hand when starting the job: table of unfinished-jobs ("bones"), finished-jobs ("done"), and jobs-fetched ("munching"). At the beginning, all the jobs are stored in the "bones" table. Each compute node ("dog") sends a request to the master to request a job from it, and starts working on it when a job is assigned by the master node. The master node then moves the fetched jobs to the "munching" table, and starts a timer for each fetched job. The timer specification will be based on the estimation of the execution time a compute node needs to finish a job. When a compute node finishes a job and notifies the master node before the job's corresponding timer runs out, this piece is regarded a finished job, and the master node moves it from the "munching" table to the "done" table. And the compute node goes on to request another job until the "bones" table is empty.
If the timer of a fetched job runs out before the compute node reports back, the master node then assumes that that particular compute node must have crashed, and then moves that timed-out job from the "munching" table back to the "bones" table for other compute nodes to fetch. The flow chart in Fig. 1 shows the master node's job handling and scheduling process (left), and the various states of the three tables (right) the master node uses to keep track of the job progress at different corresponding stages in the flow chart.
To maximize throughput of the system in a parallel environment, load balancing is required to keep the compute nodes busy by efficiently distributing the workload. The use of the "pool of tasks" is effective in achieving automatic load balancing by minimizing the idle times of compute nodes, since the nodes that finish their computations will request more tasks than the ones that require more time for their computations. This automatic, asynchronous scheduling help in keeping the compute nodes busy without having to wait for other node's computations.
The system also deals with computer nodes with heterogenous configurations. It takes advantage of the fact that each subdomain contains a different number of land points, therefore less powerful nodes can work on subdomains with less land points and more powerful ones can deal with subdomains with more land points. It has a smart scheduling in that all the compute nodes will work first on the biggest jobs they can handle, so the more powerful nodes will not waste they cycles to work on small jobs which less powerful nodes can handle, until the more powerful nodes finish the jobs other nodes can not handle.

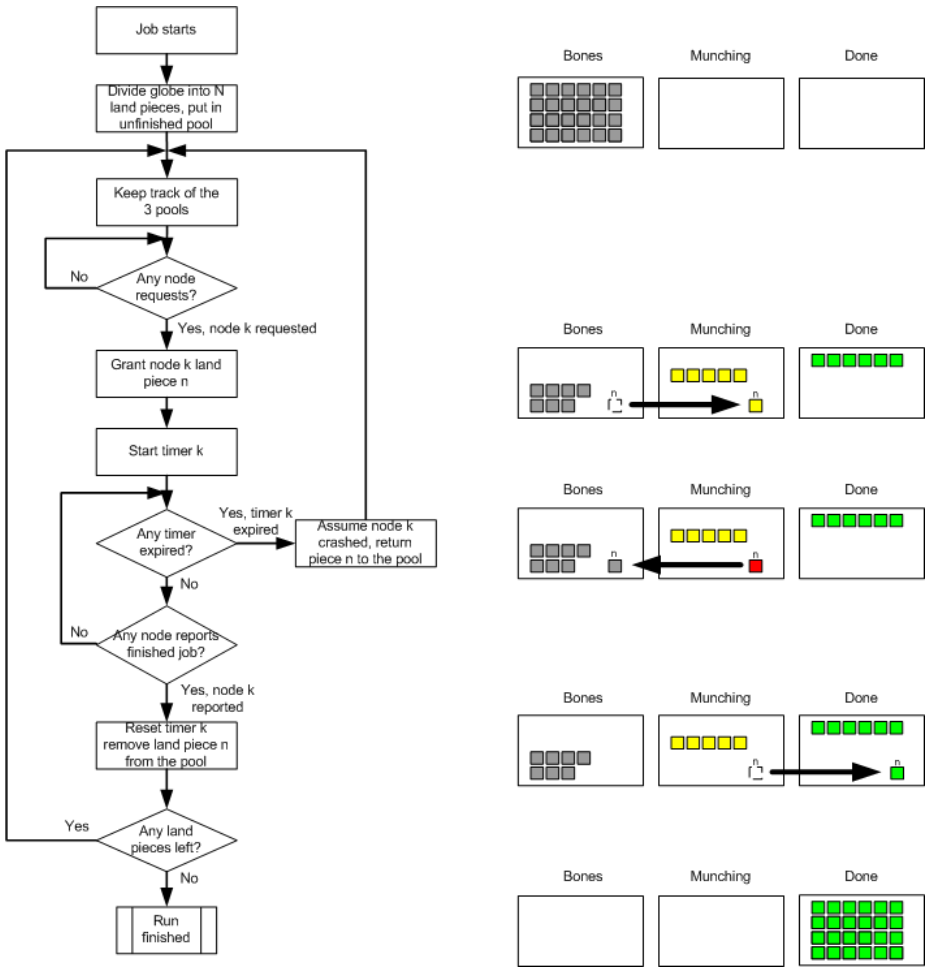

Figure 1: Task-pool-based Parallelization Scheme in LIS. Left panel shows the logic of the master node. Right panel shows the movement of jobs in the three pools: bones, munching and done. Each job in the "munching" pool is timed so when a job is timed out (red block), it will be put back to the "bones" pool for other nodes to handle. Time-out happens when a compute node crashes.

1.2 Compute node job processing

A compute node's job is to run a copy of the land surface modeling code in its process space, compute a piece of subdomain assigned by the master node, and request another piece of subdomain from the master node after it finishes the current piece, until the master node refuses to give it any pieces, in which case there are no more subdomains available and the compute node's job is done. The flow chart in Fig. 2 shows the compute node's job handling procedure and logic.

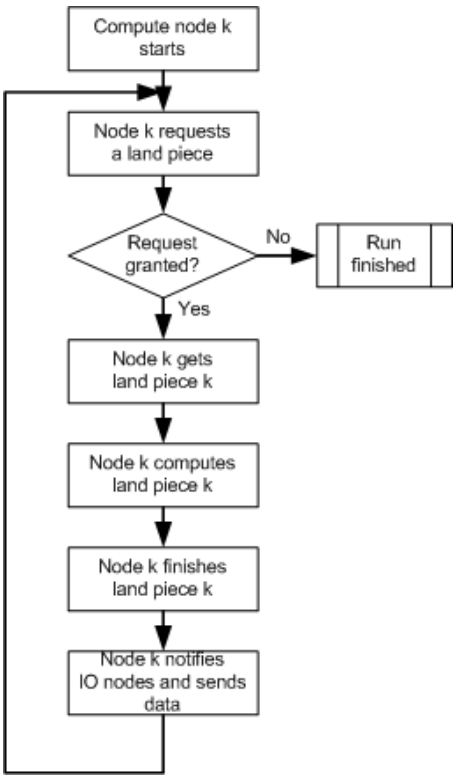

Figure 2: Flow chart of a compute node.

2 Routine/Function Prologues

2.0.1 create-bones.pl (Source File: create-bones.pl)

This program creates the task pool, after reading in the subdomain list. The location of the task pool is obtained from the config file "farmer.conf". It marks each subdomain as weight-ic_ir, where the weight is a number from 0 to 9, an indicator of computing intensity, to better utilize nodes in a heterogenous cluster where less capable nodes can deal with subdomains with less weight index and vice versa. For LIS, the weight index is selected to be in proportion of the number of land points in each subdomain.

REVISION HISTORY:

```
create-bones.pl,v 1.7 2004/03/29 22:18:27 Yudong Tian
```

USES:

```
require './getconf.pl';
```

CONTENTS:

```
# clean up the queue
$path="/data1/pool";
`rm $path/bones/*; rm $path/munching/*; rm $path/dogs/*`;
`rm $path/done/*; rm $path/logs/*; rm $path/crashes/*; rm $path/traffic_lights/*`;
```

```
my %conf = ();
&getconf (\%conf);
## get the subdomain index ic, ir from the block file
open(BLOCKS, "<$conf{blockfile}" )  or die "Can not open block file: $!\n";
@lines = <BLOCKS>;
close(BLOCKS);
$maxland = 216000;

foreach $line (@lines) {
  chomp $line;
  ($ic, $ir, $landpts) = split " ",  $line, 3;
  $ic =~ s/\s//g;  ## remove spaces
  $ir =~ s/\s//g;
  $landpts =~ s/\s//g;
  $bonew = int(($landpts-1)/$maxland*10.0);  # do not get 10
  `touch $conf{bones}/${bonew}-${ic}_${ir}`;     #
} # end of foreach
```

2.0.2 farmer.conf (Source File: farmer.conf)

Sample config file for the LIS job management, or farmer–dog system. Each entry is a single line separated by "=". Comments follow the pound sign. REVISION HISTORY:

```
$Id: farmer.conf,v 1.4 2004/03/29 22:28:10 lis Exp lis $
```

CONTENTS:

```
farmerpath = /data1/pool/control     # the location of the job management system
doglist = /data1/mpich/share/machines.LINUX        # the node list (mpich uses it too)
blockfile = /data1/pool/control/land-blocks.txt   # used to generate bones
nsegs = 1183   # number of subdomains
nnodes = 128  # number of nodes to use. nnodes + startnode < max-avail-nodes
rand_order = 1 # pick the node from the list randomly or not
startnode = 1   # which compute node to start as the first one to use
timeout = 2400   # in seconds. Max time a dog allowed to chew;

# LSM models ,experiment code and resolution
cardfile = /X8RAID/jim/run/LIS/lis.crd      # card file to back up
lsm = VIC
exp = EXP310-128
res = 1KM
snapdir = /data1/pool/snapshot   # dir to save snapshot files
```

```
# now the queues: bones, dogs, done, munching, crash
bones = /data1/pool/bones
dogs = /data1/pool/dogs
done = /data1/pool/done
munching = /data1/pool/munching

# crash management
crash = /data1/pool/crashes
recycle = 1     # send crashed job back to "bones" or not
# traffic management
traffic_lights = /data1/pool/traffic_lights
max_traffic = 180     # number of clients allowed to output simutaneously
max_wait = 30     # max number of seconds a client to wait to check traffic again

# log file location
logs = /tmp
```

2.0.3 start-farmer-dogs.pl (Source File: start-farmer-dogs.pl)

This program starts the LIS FDB (farmer-dog-bone) job management system, including "farmer.pl" and "dog.pl", on the assigned compute nodes.

REVISION HISTORY:

```
   $Id: start-farmer-dogs.pl,v 1.10 2004/03/29 22:39:25 lis Exp lis $

   !USES

 require './getconf.pl';
```

CONTENTS:

```
my %conf = ();
&getconf (\%conf);

$SIG{ALRM} = sub { die "timeout" };

$farmerpath = $conf{farmerpath};
$dognames = $conf{doglist};
$ndogs = $conf{nnodes}; # number of dogs to use
$offset = $conf{startnode}; # do not start from first node, but from somewhere
$rand_order = $conf{rand_order};  # use the nodes in random order if 1
```

```
# Start farmer on local host
print "Starting farmer.pl ... \n";
system("$farmerpath/farmer.pl >&/tmp/farmer.out &");

open(DOGS, "<$dognames") or die "Can not get the dogs\' name list: $!";
@lines = <DOGS>;
close(DOGS);

if($rand_order) {
  fisher_yates_shuffle( \@lines );
}

# do not go beyond max nodes we have
$min = scalar(@lines) < ( $ndogs + $offset ) ? scalar(@lines):($ndogs + $offset);
print "nodes limit: $min\n";
for (my $i=$offset; $i < $min; $i++) {
   $dog = $lines[$i];
   chomp $dog;
   if ($dog) { # If not empty line
    print "Starting dog $dog ... \n";
    eval {
     alarm(10);     # 10 seconds time out
     $output=`ssh -4 $dog "$farmerpath/dog.pl >& /tmp/dog.out &" `;
     alarm(0);
    };  # end eval
    print $output;
   } # end if
} # end for


    # fisher_yates_shuffle( \@array ) :
    # generate a random permutation of @array in place
    sub fisher_yates_shuffle {
        my $array = shift;
        my $i;
        for ($i = @$array; --$i; ) {
            my $j = int rand ($i+1);
            next if $i == $j;
            @$array[$i,$j] = @$array[$j,$i];
        }
    }
```

2.0.4 farmer.pl (Source File: farmer.pl)

This is the "farmer" process in the LIS FDB (farmer-dog-bone) job management system. This program is repsonsible for giving out tasks to compute nodes ("dogs"), and for taking tasks back if a node crashes, or it can not finish timely.

REVISION HISTORY:

```
$Id: farmer.pl,v 1.16 2004/03/25 15:37:11 lis Exp lis $
```

USES:

```
require './getconf.pl';
```

CONTENTS:

```
my %conf = ();
&getconf (\%conf);

$Sleeptime = 4;  # in seconds. Take a short break for the farmer.
$MaxMunchTime = $conf{timeout};  # in seconds. Max time a dog allowed to chew.
$Bones = $conf{bones};
$Dogs =  $conf{dogs};
$Munching = $conf{munching};
$Done = $conf{done};
$Flog = "$conf{logs}/farmer.log";
$Traffic_lights = $conf{traffic_lights};
# create snapshot dir
$Snapdir = "$conf{snapdir}/$conf{res}/$conf{lsm}/$conf{exp}/";
$Cardfile = $conf{cardfile};
`mkdir -p $Snapdir`;
`cp $Cardfile $Snapdir`;   # back up a copy of the card file

open(LOG, ">> $Flog") or die "Can not open log file: $!";
$| = 1;

print LOG scalar localtime, ":Starting farmer.pl ... \n";

@munchings = `ls -U $Munching`;
@bones = `ls -U $Bones |/bin/sort -n -r`;

# keep looking after the business until no bones left, and no dogs munching
while ( scalar (@munchings) > 0 || scalar (@bones) > 0  ) {

# Check if there are expired munchings

@munchings = `ls -U $Munching`;
#print LOG scalar localtime, ":There are ", scalar(@bones), " bones waiting.\n";
#print LOG scalar localtime, ":There are ", scalar(@munchings), " dogs munching.\n";
```

```
foreach $munch (@munchings) {
  chomp $munch;
  $systime = `date +%s`;
  chomp $systime;
  $filetime = `date -r $Munching/$munch +%s`;
  # print "$Munching/$munch \n";
  chomp $filetime;
  # print $systime - $filetime, "\n";
  if ($systime - $filetime > $MaxMunchTime ) { # That dog must have crashed
     # put back the bone
     ($dogfile, $bonefile) = split /-/, $munch, 2;
     `mv $Munching/$munch $Bones/$bonefile`;
     unlink "$Traffic_lights/$host";  # remove from traffic lane
     print LOG scalar localtime, ":Dog $dogfile expired. Bone $bonefile back.\n";
  } # end if. Do nothing if not expired
}  # end for

# Check if there are any dogs barking

@dogs = `ls -U $Dogs`;
@bones = `ls -U $Bones |/bin/sort -n -r`;
$ndogs = scalar(@dogs);
$nbones = scalar(@bones);
if ($ndogs) {
  print LOG scalar localtime, ":There are $ndogs dogs barking ...\n";
  if (scalar(@bones)) {
    print LOG scalar localtime, ":There are $nbones bones left ...\n";
    # find a bone with the right size to match each dog
    foreach $dog (@dogs) {   # dog form: hostname-maxweight, eg. A3-8
      chomp $dog;
      ($host, $dogw) = split /-/, $dog, 2; # get the host & weight
BONES: for (my $i=0; $i < $nbones; $i++) {
        chomp $bones[$i];
        ($bonew, $block) = split /-/, $bones[$i], 2; # get bone weight
        if ($bonew <= $dogw) { # find the right bone for the dog. Bone form: 8-12_32
          unlink "$Dogs/$dog";
          `cp $Bones/$bones[$i] $Munching/$host-$bonew-$block`;
          unlink "$Bones/$bones[$i]";
          splice @bones, $i, 1;  # remove from the list
          print LOG scalar localtime, ":Dog $dog given bone $bones[$i].\n";
          last BONES;
        } # end if
       } # end BONES loop
     }   # end foreach dog
  } # end if bones
} # end if ndogs: finish giving out the bones
```

```
sleep $Sleeptime;  # sleep a while if no dogs to feed
@munchings = `ls -U $Munching`;
@bones = `ls -U $Bones |/bin/sort -n -r`;

# save snap shots
`ls -U $Bones > $Snapdir/bones.txt`;
`ls -U $Done > $Snapdir/done.txt`;
`ls -U $Munching > $Snapdir/munching.txt`;
} #  End while

print LOG scalar localtime, ":Job done ... \n";
close(LOG);
`mv $Flog $Snapdir`;  # save the log file


%////////////////////////////////////////////////////////////
{\sf CONTENTS:}
 \begin{verbatim}

\markboth{Left}{Source File: dog.pl,  Date: Tue Apr 13 16:07:44 EDT 2004
}

#!/usr/bin/perl
#
```

2.0.5 dog.pl (Source File: dog.pl)

This program runs on compute nodes ("dogs") and is responsible for asking for and get tasks ("bones") from the master node ("farmer") to the compute nodes. Then it starts LIS code to work on the "bones". It will check the compute node's memory and determine the maximum weight of the subdomains it can handle. It also has the option to recycle a crashed subdomain or not. It is started by "start-farmer-dogs.pl".

REVISION HISTORY:

```
    dog.pl,v 1.21 2004/03/29 22:33:23 Yudong Tian
```

USES:

```
 require '/data1/pool/control/getconf.pl';
```

CONTENTS:

```
my %conf = ();
&getconf (\%conf);
```

```
#‘/bin/rm -rf /tmp/*‘;  # clean tmp
#‘/bin/rm -rf ~/.dods_cache‘;  # clean dods client cache
$Sleeptime = 10;  # in seconds. Take a short break for the dog after it does a job.
$Bones = $conf{bones};
$Dogs = $conf{dogs};
$Munching = $conf{munching};
$Done = $conf{done};
$Crash = $conf{crash};
$Logs = $conf{logs};
$Recycle = $conf{recycle};
$Traffic_lights = $conf{traffic_lights};

# Determine dog weight
$mem = ‘free -t |tail -1 |awk ’{print \$2}’‘;  # get total RAM
chomp $mem;
$dogw = 4;   # clm: 4. Noah: 9; vic : 5
$dogw = 9 if ($mem > 600000);   #if 1GB RAM, ask for bigger bones

$hostname = ‘/bin/hostname‘;
chomp $hostname;
($host, $rest) = split /\./, $hostname;
$host =~ tr/a-z/A-Z/;
open(LOG, ">> $Logs/$host.log") or die "Can not open log file: $!";
$| = 1;

print LOG scalar localtime, ":Dog $host started ... \n";

sleep (int(rand 240)); # slow start

@munchings = ‘ls -U $Munching‘;
@bones = ‘ls -U $Bones‘;

# Staying around until no bones left, and no dogs munching
while ( scalar (@munchings) > 0 || scalar (@bones) > 0  ) {

# Check if given a bone

$mybone = ‘ls $Munching/$host-*‘;
chomp $mybone;
if ($mybone) { # got somthing to do now
  ($path, $bonew, $boneid) = split /-/, $mybone, 3;  # bones/A3-8-23_13
  print LOG scalar localtime, ":I got $mybone\n";
  # start munching
  {
   ($ic, $ir) = split /_/, $boneid, 2;
   #‘/data1/test-I/LDAS/pc.noah.geos.1km.sh 1 $ic $ir‘;
   #####- The real job: here--------------------#####
   ‘/X8RAID/jim/run/LIS/1km.sh 1 $ic $ir‘;
```

```
   #####----------------------------------------#####
   $enum = $?;  # record error number
  }
   $mybone = `ls $Munching/$host-*`;  # Return the full path
   chomp $mybone;
   if ($mybone) { # ha! it is still there. I was munching faster enough
     #`mv $Munching/$mybone $Done/$mybone`;
     ($path, $boneid) = split /-/, $mybone, 2;  # A3-8-23_23 => 8-23_23
     #`mv $mybone $Done/$boneid`;
     if( $enum == 0 ) {   # job done. See if my assignment is still there
      `mv $mybone $Done/`;   # keep the hostname there for debugging
       print LOG scalar localtime, ":I got $mybone done!\n";
     } else { # crashed. move to crashed pile or back to bones
       print LOG scalar localtime, ": $mybone crashed!\n";
       if ($Recycle) {
        `cp $mybone $Crash/`;   # keep the hostname there for debugging
        `mv $mybone $Bones/$boneid`;
         print LOG scalar localtime, ": $mybone recycled as $boneid!\n";
       } else {
       `mv $mybone $Crash/`;   # keep the hostname there for debugging
       }
       unlink "$Traffic_lights/$host";
       sleep $Sleeptime;  # in seconds. Take a short break
     } # end if enum
   } # end if. Otherwise, just do nothing. :-<
} # end if. If not given a bone, bark follows below

unless ( -e "$Dogs/$host-$dogw" or `ls $Munching/$host-*`) {
  `touch $Dogs/$host-$dogw`;
  print LOG scalar localtime, ":I am barking now!\n";
}   #

sleep $Sleeptime;  # in seconds. Take a short break

@munchings = `ls -U $Munching`;
@bones = `ls -U $Bones`;

} # end while. No bones left, no dogs munching.

# erease my bark if any left answered.
unlink "$Dogs/$host-$dogw";
print LOG scalar localtime, ":I am all done!\n";
close(LOG);
```

2.0.6 getconf.pl (Source File: getconf.pl)

This perl subroutine reads a config file "/data1/pool/control/farmer.conf" and parses the entries. It returns the entries in a hash. Usage example:

```
%conf = ();
  &getconf (\%conf);
foreach $key (keys %conf) {
   print "$key=$conf{$key}\n";
}
```

REVISION HISTORY:

```
$Id$
```

CONTENTS:

```
sub getconf {
 my @lines, $line, $key, $value;
 my $pconf = shift;
 open(CONF, "</data1/pool/control/farmer.conf") or die "Can not open conf file: $!\n";
 @lines = <CONF>;
 close(CONF);
 foreach $line (@lines) {
   chomp $line;
   $line =~ s/\#(.*)$//;  # remove comments
   $line =~ s/\s//g;  # remove spaces
   if ($line) {  # if anything left ...
     ($key, $value) = split /=/, $line, 2;
      $$pconf{$key} = $value
   } # end if
 } # end of foreach

}  # end of getconf

1;
```